

Наследяване (Rev: 1.2)

Любомир Чорбаджиев¹
lchorbadjiev@elsys-bg.org

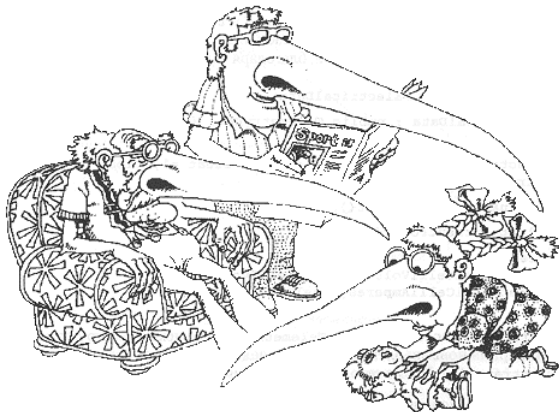
¹Технологическо училище “Електронни системи”
Технически университет, София

1 февруари 2007 г.

Съдържание

- 1 Наследяване
- 2 Наследяване в C++
- 3 Полиморфизъм

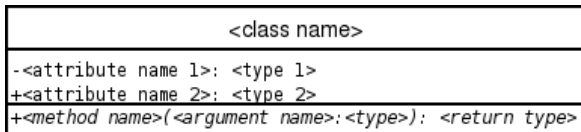
Наследяване



¹Grady Booch, *Object Oriented Analysis and Design with Applications*, 1991.

UML

- UML – unified modeling language.
- Използва се за моделиране на обектно ориентирани програмни системи.
- Дефинира набор от различни видове диаграми.
- Най-често използваните диаграми са “клас диаграмите”.



Клас Employee

Employee
-name_: string
-id : long
+get_name(): string
+get_id(): long

```

1 class Employee {
2     string name_;
3     long id_;
4 public:
5     Employee(string name, long id)
6         : name_(name), id_(id)
7     {}
8     string get_name() const;
9     long get_id() const;
10 };

```

Клас Manager

Manager
-name_: string
-id_: long
-level_: int
+get_name(): string
+get_id(): long
+get_level(): int

```

1 class Manager {
2     string name_;
3     long id_;
4     int level_;
5 public:
6     Manager(string name,
7             long id,
8             int level)
9         :name_(name), id_(id),
10        level_(level)
11    {}
12    string get_name() const;
13    long get_id() const;
14    int get_level() const;
15 };

```

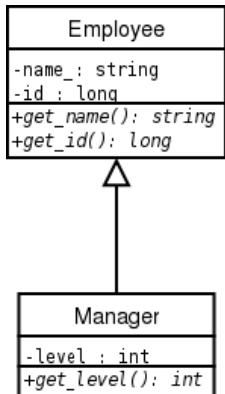
Класовете Employee и Manager

Employee
-name_ : string
-id : long
+get_name(): string
+get_id(): long

Manager
-name_ : string
-id_ : long
-level : int
+get_name(): string
+get_id(): long
+get_level(): int

- Мениджърът (Manager) е вид работник (Employee).
- Класът Manager притежава всички атрибути и методи на класа Employee.
- Освен атрибутите и методите на класа Employee, класът Manager притежава някои допълнителни свойства:
 - ниво в йерархията (level_);
 - група от подчинени;
 - ...

Наследяване



- За да се моделира подобен вид отношение между класовете в обектно-ориентираното програмиране се използва наследяване.
- Класът `Employee` се нарича **базов клас** или **супер клас**.
- Класът `Manager` се нарича **производен клас** или **наследник**.

Наследяване в C++

```
1 class Employee {  
2     string name_;  
3     long id_;  
4 public:  
5     Employee(string name, long id)  
6     : name_(name), id_(id)  
7     {}  
8     string get_name() const;  
9     long get_id() const;  
10 };
```

Наследяване в C++

```
1 class Manager: public Employee {
2     int level_;
3 public:
4     Manager(string name, long id, int level)
5     : Employee(name, id),
6       level_(level)
7     {}
8     int get_level(void) const;
9 };
```

- Повторно използване на кода (code reuse).
 - Методите `get_name()` и `get_id()` са дефинирани само веднъж — в класа `Employee`.
 - Класът `Manager` също има методи `get_name()` и `get_id()`, наследени от `Employee`.

Наследяване в C++

```
1 int main() {
2 Employee e1("Иван_Работников", 8101011);
3 Manager m1("Шеф_Иванов", 8012121, 1);
4 //...
5 Employee* employee_list[10];
6 employee_list[0]=&e1;
7 employee_list[1]=&m1;
8 //...
9 Manager* manager_list[10];
10 manager_list[0]=&m1;
11 manager_list[1]=&e1; // error!!!
12
13 return 0;
14 }
```

Наследяване в C++

- Наследяването на `Manager` от `Employee` превръща `Manager` в подтип на `Employee`.
- `Manager` е `Employee` и поради това `Manager*` може да се използва навсякъде, където трябва да се използва `Employee*`.
- Обратното, обаче, не е вярно; не всеки `Employee` е `Manager` и следователно `Employee*` не може да замести използването на `Manager*`.

Наследяване в C++

```
1 class Employee;  
2  
3 class Manager: public Employee { // error!!!  
4     //...  
5 };
```

- За да може един клас да се използва като базов клас, той трябва да е дефиниран и неговата дефиниция да е видима при декларирането на производния клас.
- В ред 3 е допусната грешка: класът `Employee` само е деклариран, но неговата дефиниция липсва.

Методи

```
1 class Employee {  
2     string name_;  
3     long id_;  
4 public:  
5     Employee(string name, long id)  
6         : name_(name), id_(id)  
7         {}  
8     string get_name(void) const;  
9     long get_id(void) const;  
10    void print(void) const;  
11 };
```

Методи

```
1 class Manager: public Employee {  
2     int level_;  
3 public:  
4     Manager(string name, long id, int level)  
5         : Employee(name, id),  
6           level_(level)  
7     {}  
8     int get_level(void) const;  
9     void print(void) const;  
10 };
```

Методи

- Ще разгледаме дефиницията на метода `print()` в класа `Manager`.
- Методите на класа наследник имат достъп до публичните (`public`) и защитените (`protected`) член-променливи и методи на базовият клас

```
1 void Manager::print(void) const {  
2     cout << "Name: " << get_name() << endl;  
3 }
```

- Методите на класа наследник **нямат** достъп до скритите (`private`) член-променливи и методи на базовият клас

```
1 void Manager::print(void) const {  
2     cout << "Name: " << name_ << endl; // грешка!!  
3 };
```


Методи

- При реализацията на производен клас трябва да се използват само публичните методи и член-променливи на базовия клас.
- Като компромис могат да се използват защитени (**protected**) член-променливи и методи.
- Защитените членове на базовия клас се държат като публични (**public**) за членовете на производните класове и като скрити (**private**) за всички останали функции.

- В базовия и производния класове може да се дефинират методи с еднакви имена.
- В класовете `Employee` и `Manager` е дефиниран метод с едно и също име — `print()`.
- Когато в класа наследник искаме да използваме метода `print()`, дефиниран в базовият клас, трябва да използваме пълното му име.

```

1 void Manager::print(void) const {
2     Employee::print();
3     //...
4 }

```

- Следната дефиниция е некоректна (безкрайна рекурсия):

```

1 void Manager::print(void) const {
2     print();
3     //...
4 }

```

Конструктори

- При създаване обект от производен клас: първо се създава базовият клас, след това се създават член-променливите на производния клас и след това самият производен клас.

```
1 class Manager: public Employee {  
2     int level_;  
3 public:  
4     Manager(string name, long id, int level)  
5     : Employee(name, id),  
6       level_(level)  
7     {}  
8     ...  
9 };
```

Конструктори

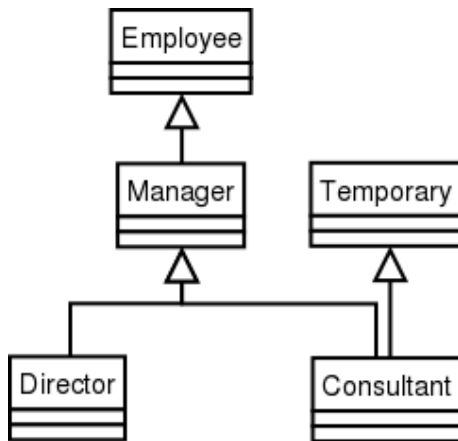
- В конструктора на производния клас могат да се инициализират член-променливите, които са декларирани в производния клас.
- Не е възможно да се инициализират директно член-променливите на базовия клас.
- Ако базовият клас няма конструктор по подразбиране, то в производния клас задължително трябва да се извика конкретен конструктор на базовият клас.
- Ако в производния клас не е изрично указано кой конструктор на базовия клас трябва да се извика, то ще се извика конструкторът по подразбиране на базовия клас.

Конструктори

```
1 class Manager:public Employee {  
2 //...  
3 };  
4 Manager::Manager(string name, long id,  
5                 int level)  
6     : name_(name),  
7       id_(id),  
8       level_(level)  
9 {...}
```

Колко са грешките в дефиницията на конструктора на класа Manager?

Йерархия от класове



Йерархия от класове

```
1 class Employee{/*...*/};
2 class Temporary{/*...*/};
3 class Manager:public Employee {/*...*/};
4 class Director: public Manager {/*...*/};
5 class Consultant: public Manager,
6                   public Temporary{
7   /*...*/};
```

Заместване на функции

```
1 class Employee {  
2     ...  
3 public:  
4     ...  
5     void print(void) const;  
6 };
```

```
1 class Manager: public Employee {  
2     ...  
3 public:  
4     ...  
5     void print(void) const;  
6 };
```


Заместване на функции

```
1 Employee e1("Иван_Работников", 8101011);
2 Manager m1("Шеф_Иванов", 8012121, 1);
3 //...
4 vector<Employee*> employee_list;
5 employee_list.push_back(&e1);
6 employee_list.push_back(&m1);
7 //...
8 employee_list[0]->print();
9 employee_list[1]->print();
```

Кой метод се извиква в ред 9? Employee::print() или Manager::print()?

Заместване на функции

- Трябва да се дефинира функция, която да разпечатва на стандартния изход данните за всички работници.
- Нека разгледаме следната примерна реализация:

```
1 vector<Employee*> v;  
2 ...  
3 for (vector<Employee*>::iterator it=v.begin();  
4     it!=v.end(); ++it){  
5     (*it)->print();  
6 }
```

- В тази функция в ред 5 винаги се извиква функцията `Employee::print()`. Такава реализация е неудовлетворителна, тъй като се губи спецификата на различните видове работници.

Член-променлива за типа

- Вариант за решаването на проблема е в класа `Employee` да се добави член-променлива, в която се помни типа на обекта.

```
1 class Employee {  
2 public :  
3     enum EmployeeType {E, M};  
4     EmployeeType employee_type;  
5 private :  
6     ...
```

Член-променлива за типа

```
1 public :  
2     Employee(string name, long id)  
3     : employee_type(E),  
4       name_(name),  
5       id_(id)  
6     {}  
7     ...  
8 };
```

Член-променлива за типа

```
1 class Manager:public Employee {
2     ...
3 public:
4     Manager(string name, long id, int level)
5     : Employee(name, id),
6       level_(level)
7     {
8         employee_type=Employee::M;
9     }
10 };
```

Член-променлива за типа

```
1 void print_all(vector<Employee*>& v){
2     for(vector<Employee*>::iterator
3         it=v.begin();
4         it!=v.end();++it){
5         if((*it)->employee_type==Employee::M){
6             // print manager
7         }
8         else {
9             // print employee
10        }
11    }
12 }
```

Член-променлива за типа

- Такова решение на проблема може да работи в малка програма, но когато йерархията от класове нараства, броят на проверките за типа на променливата също нараства.
- Когато се добавя нов клас в йерархията трябва да се променят всички функции, които зависят от проверки за типа.
- Когато се добавя нов клас в йерархията трябва да се промени и базовият клас.
- Използването на член-променлива за типа противоречи на идеята за капсулиране на данните.

Виртуални функции

```
1 class Employee {
2     string name_;
3     long id_;
4 public:
5     Employee(string name, long id)
6         : name_(name),
7           id_(id)
8     {}
9     virtual void print(void) const{
10         cout << name_ << "\n"
11             << id_ << endl;
12     }
13 };
```


Виртуални функции

```
1 class Manager:public Employee {
2     int level_;
3 public:
4     Manager(string name, long id, int level)
5         : Employee(name, id),
6           level_(level)
7     {}
8     void print(void) const {
9         Employee::print();
10        cout << "\tlevel:" << level_ << endl;;
11    }
12};
```

Виртуални функции

```
1 void print_all(vector<Employee*>& v){
2     for(vector<Employee*>::iterator
3         it=v.begin();
4         it!=v.end();++it){
5         (*it)->print();
6     }
7 }
```

Виртуални функции

```
1 int main(int argc, char* argv){
2     Employee e("Brown", 81010110L);
3     Manager m("Smith", 80121212L, 1);
4
5     vector<Employee*> employees;
6     employees.push_back(&e);
7     employees.push_back(&m);
8
9     print_all(employees);
10    return 0;
11 }
```

Виртуални функции

- Функцията `Manager::print()` замества (override) функцията в базовия клас `Employee::print()`;
- Функцията в базовия клас е дефинирана като виртуална. Това означава, че изборът, коя функция да се извика се определя по време на изпълнение на програмата, в зависимост от действителния тип на променливата.

```
1 void print_all(vector<Employee*>& v){
2     for (vector<Employee*>::iterator
3         it=v.begin();
4         it!=v.end();++it){
5         (*it)->print();
6     }
7 }
```

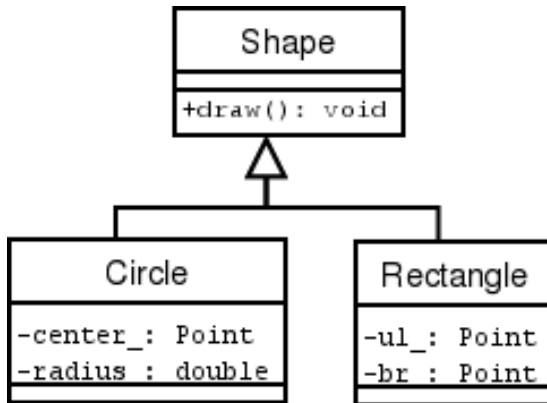
Виртуални функции

- Когато функция от производния клас, има същата сигнатура като виртуална функция в базовия клас, се казва че тя **замества (override)** виртуалната функция от базовия клас.
- Когато се извиква виртуална функция, то автоматично се използва най-подходящият ѝ заместник, в зависимост от действителния тип на извикващия обект.

Виртуални функции

- Поведението, при което конкретната функция, която се извиква, зависи от динамичния тип на обект, чрез който е извикана, се нарича **полиморфизъм** или **динамично свързване**.
- За да бъде една член-функция полиморфна в C++ е необходимо тя да се декларира като виртуална с помощта на модификатора **virtual**.

Абстрактни класове



Абстрактни класове

```
1 class Shape {  
2 public:  
3     virtual void draw(void) const=0;  
4 };
```

- В ред 3 е дефинирана **чисто виртуална** функция.
- Класове, в които са дефинирани една или повече чисто виртуални функции се наричат **абстрактни**.

Абстрактни класове

- Не е възможно да се създаде обект от абстрактен клас:

```
Shape s; //error!!!
```

- Абстрактните класове се използват за дефиниране на интерфейса на йерархия от класове.
- Чисто виртуалните функции, които не са определени в производните класове остават чисто виртуални. Това означава, че е възможно и базовият и производният клас да бъдат абстрактни.

Абстрактни класове

```
1 class Shape {
2 public:
3     virtual void draw(void) const=0;
4 };
5
6 class Circle: public Shape {
7     Point center_;
8     int radius_;
9 public:
10    Circle(const Point& center, int radius)
11        : center_(center), radius_(radius)
12    {}
```

Абстрактни класове

```
1 void draw(void) const {
2     cout << "Circle::draw(";
3     center_.print();
4     cout << ",□" << radius_ << ")" << endl;
5 }
6 };
7 class Rectangle: public Shape {
8     Point ul_;
9     Point br_;
10 public:
```

Абстрактни класове

```
1 Rectangle(const Point& ul, const Point& br)
2     : ul_(ul), br_(br)
3 {}
4 void draw(void) const {
5     cout << "Rectangle::draw(";
6     ul_.print();
7     cout << ",␣";
8     br_.print();
9     cout << ")" << endl;
10 };
11 };
```

Абстрактни класове

```
1 class Drawing {
2     list<Shape*> shapes_;
3 public:
4     ...
5     void draw(void) const {
6         for(list<Shape*>::const_iterator
7             it=shapes_.begin();
8             it!=shapes_.end();++it){
9             (*it)->draw();
10        }
11    };
12};
```

Динамично преобразуване

- C++ поддържа RTTI — идентификация на типа по време на изпълнение.
- RTTI ни дава възможност да идентифицираме истинския тип, към който сочи даден указател.
- RTTI може да се използва по няколко начина — единият е динамичното преобразуване на типовете **dynamic_cast**<...>(...).

```
1 void fun(Shape* sh){  
2     Circle* pc=dynamic_cast<Circle*>(sh);  
3     ...  
4 }
```

- Когато преобразуването е успешно операторът за динамично преобразуване на типа връща ненулев указател.

Динамично преобразуване

```
1 void fun(Shape* sh){
2   Circle* pc=dynamic_cast<Circle*>(sh);
3   if(pc!=NULL){
4     // it's a circle
5     return ;
6   }
7   Rectangle* pr=dynamic_cast<Rectangle*>(sh);
8   if(pr!=NULL){
9     // it's a rectangle
10  }
11 }
```